

EXHIBIT 27

VMTP: A Transport Protocol for the Next Generation of Communication Systems

David R. Cheriton
Computer Science Department
Stanford University

Abstract

The Versatile Message Transaction Protocol (VMTP) is a transport-level protocol designed to support remote procedure call, multicast and real-time communication. The protocol is optimized for efficient page-level network file access in particular.

In this paper, we describe the significant aspects of the VMTP design, including the VMTP treatment of sessions, addressing, duplicate suppression, flow control and retransmissions plus its provision for multicast. The VMTP design reflects a change in the use of computer communication as well as a change in the underlying hardware base for the next generation of communication systems. It also challenges certain established notions in the design of protocols.

1 Introduction

Computer communication use has been dominated, until recently, by remote terminal access and file transfer, basically *inter-system* communication.¹ Transport protocols have been designed with the characteristics of these application in mind, leading to virtual circuit-based protocols. With the development of distributed systems, use is shifting significantly to *intra-system* communication such as remote procedure call, multicast and real-time datagrams. Remote procedure call (RPC)[1] arises with distributed programming in general. A specific case of RPC, page-level network file access, is a performance-critical intra-system communication function, especially with the increasing use of shared network file servers and diskless workstations. Multicast has been recognized as a powerful facility in distributed systems for implementing decentralized naming[6], distributed scheduling, parallel computation[7], distributed transaction management and replication[10]. Finally, use of clusters of machines for real-time process control, real-time conferencing and data collection is growing. These uses are not well served by current transport protocols.

The common communication substrate has been, until recently, wide-area networks based on telephone technology and low performance computer switching nodes. Communication bandwidth, node buffer space and node processing power have been the critical resources. A significant shift is taking place to local area networks with multi-megabit data rates, low delay and low error rate. In addition, the future promises high-speed, low error rate, wide-area fiber optic channels plus high-performance switching nodes and gateways that take advantage of the low cost of memory, processors and multi-processor technology. These advances suggest the addition of new wide-area services such as internetwork multicast[5]. They also suggest rethinking the design of protocols. We claim that current transport protocols are not well suited to the new *communication economics* introduced by this shift in communication substrate.

The Versatile Message Transaction Protocol (VMTP) is a transport-level protocol designed in response to these shifts in communication usage and substrate. In this paper, we describe the significant aspects of the VMTP design, including the VMTP treatment of sessions, addressing, duplicate suppression, flow control and retransmissions plus its provision for multicast. The VMTP design reflects a change in the use of computer communication and the change in the underlying hardware base for the next generation of communication systems. It also challenges certain established notions in the design of protocols.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

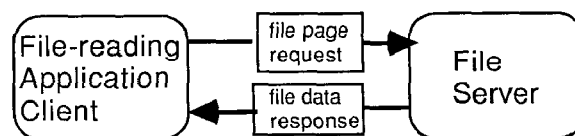


Figure 1: Basic VMTP Message Transaction: Network File Read

The next section describes key aspects of the VMTP design. Section 3 describes details of the protocol, including packet format. Section 4 indicates the current status of our implementation and experimentation with performance. Section 5 compares this protocol to other related work. We close with a summary of the key points plus an indication of future directions.

2 Key Design Aspects of VMTP

VMTP is designed for use by a group of computers operating as a distributed system. Inter-node communication is assumed to be dominated by page-level file access, application-level remote procedure calls, real-time datagrams and multicasts. File transfer and remote terminal access are subsumed as special cases of file access. The participating computers may be distributed across an internetwork such as the DoD Internet. However, we assume that statistically, most communication is local to a local network or tightly-coupled cluster of local networks and optimized accordingly. Moreover, we assume that the chasm between local network performance and wide-area performance will be reduced with the increasing use of the new substrate describe earlier. In particular, we assume a datagram multicast facility such as available on the Ethernet and being developed for the Internet[5].

VMTP is basically a request-response protocol. A VMTP session, a *message transaction*, is initiated by a *client* sending a *request message* to a *server* entity and terminated by the server sending back a *response message*. The response acknowledges to the client receipt of the request message. The next request from the client, explicit acknowledgement or timeout acknowledges to the server receipt of the response by the client. The basic VMTP message transaction is illustrated in Figure 1 in an expected common use, network file page read. A client can only have one message transaction outstanding at one time although a host may implement multiple VMTP clients. Request and response may consist of multiple packets. A VMTP message transaction takes place between network-visible *entities*, client entities and server entities. An entity may be (for example) a process, port or procedure invocation. Each entity is addressed by a 64-bit *entity identifier*. A group of entities can be identified by a single entity identifier even if they are distributed across several machines. For example, a single identifier can specify the group of file servers. We do not prohibit an entity from having several different entity identifiers, although the protocol has no notion of different entity identifiers specifying the same entity.

The following subsections describe key aspects of VMTP in detail and justify the design in terms of these assumptions about the use and substrate.

¹Here, *system* is used in the sense of an autonomous computer system.

2.1 Conversation Support

A transport protocol must support conversations, where a *conversation* is a sequence of related communication actions. Examples include the open-read-write-close operations in a single open file, terminal read-write actions in a session and the sequence of data transfers that constitute a file transfer. We use the term *connection* to refer to a protocol mechanism for implementing conversations. That is, a connection is a logical entity in a protocol that associates a set of communication actions.

Most transport-level protocols use the virtual circuit notion of *connection* to support user-level conversations. That is, the user of the transport protocol maps its notion of conversation onto a transport-level connection or virtual circuit. File transfer and remote terminal access are two prime examples. However, we claim that: *Most communication in distributed systems either requires a conversation at a higher level than the transport level or does not require a conversation.* Thus, invoking a version of the end-to-end argument[13], a transport-level connection can only duplicate the higher level connection actions and cannot replace the need for this connection. Moreover, there are no significant performance benefits to a transport-level connection in our assumed communication environment.

In contrast, VMTP provides facilities for the higher level modules to implement conversations but does not implement conversations directly. The two key aspects to VMTP conversation support are: stable addressing and message transactions. A *stable address* is one that retains the same "meaning" or binding as long as it remains valid. A *message transaction* is a request-response pair with reliable delivery on both the request and the response messages.

A conversation is typically structured on top of VMTP message transactions as a sequence of message transactions addressed to the same server address and server-specific conversation or connection identifier.² Because of stability, the client knows that each request message is delivered to the same server. Using the common conversation identifier and client address, the server associates each request message with the same conversation and authorization. Each response is associated with the conversation by its association with a request message that is part of the conversation. In addition, the client and server can check the existence of the other using the other's address, again relying on stability.

Use of these facilities is illustrated by considering open file connections. The client sends an *open* request to the file server. The file server opens the file (assuming file protections are satisfied) and allocates an open file descriptor for file. It records the client address, permissions and other relevant information in this descriptor, allocates and records a local open file identifier and returns this information to the client. Subsequent client requests specify this open file identifier, allowing the server to map to the particular open file descriptor and check the client address against the one previously recorded. Reliable delivery and duplicate suppression for request and response messages are handled by VMTP.

Conversation support in VMTP is preferred over directly implementing connections, as in TCP, for several reasons:

- **Minimal Redundancy:** Open file connections require file-related state. Implementing connections or virtual circuits at the transport level does not eliminate the requirement for file-level connections. Thus, to avoid duplication of mechanism, VMTP does not itself implement conversations.
- **Minimal Server State:** A server has at most one VMTP record for each active client, independent of the number of higher level conversations the client has with the server. This record, called *transaction record*, is reused if a new request is received from the client within a timeout period and automatically released otherwise. A transaction record is automatically allocated for the client if one is not allocated on receipt of a new request. Thus, on a conversation of reasonable activity, the transaction record is present when the next request message from the same client arrives so, statistically, it behaves like a connection descriptor. However, the automatic allocation and reclamation obviates the

²Services that do not require conversation support can be handled as degenerate conversations.

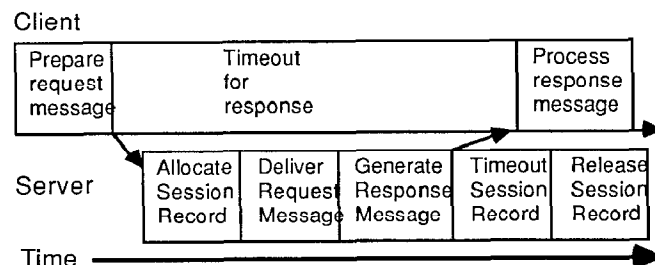


Figure 2: Client/Server Transaction Actions

need for explicit open and close packets at the transport level. The client and server actions as part of a transaction are illustrated in Figure 2

- **Minimal client state:** Each client requires only one VMTP record, independent of the number of conversations the client has. It simply reuses the descriptor on each message transaction, taking advantage of the fact that a client may only have one outstanding transaction at a time.
- **Flexible higher-level conversations:** A VMTP user can construct a variety of different types of conversations including multicast and real-time message transactions, as described in the next section. For example, a monitoring device can have a conversation with logging devices using datagram multicast to the "log device group". The association of communication actions with the conversation may be based on the multicast address, monitoring-specific information and timestamps. Moreover, the log devices, as VMTP servers, may be read by user query programs using normal VMTP transactions³.

In contrast to VMTP message transactions, using a virtual circuit mechanism for file access implies either a client has a virtual circuit per open file or multiplexes multiple open file connections on a single transport-level virtual circuit. In the former case, the number of virtual circuits required can become prohibitively expensive, both in the setup, management and tear-down time as well as the space for connection records. For instance, it is not uncommon for a compiler to have 10 files open simultaneously.

In the latter case, there is considerable complexity and overhead in connection management if multiple open file connections are multiplexed on a single virtual circuit. For instance, the flow control and reliable delivery of a virtual circuit implies that either an endpoint must read and retain all data available on the virtual circuit or else unrelated messages may be blocked from being received by flow control. In either case, a virtual circuit-based implementation does not extend well to multicast and datagram communication.

The redundancy of transport-level connections relative to higher-level connections was less a disadvantage in previous communication systems because remote terminal access and file transfer, the main two applications, typically only used one connection. However, with applications like compilers, editors and formatters that have many open files simultaneously, this redundancy becomes a significant issue for page-level file access.

As an aside, this discussion is not directly part of the virtual circuit versus datagram debate. That debate focuses on basic delivery model of the network or internetwork level. We focus on the transport level, which is defined as providing reliable delivery. In this vein, we argue for the transaction model as a degenerate, and therefore less expensive, form of virtual circuit.⁴ However, VMTP is defined in terms of a datagram delivery model of the network or internetwork.

In summary, the message transaction model provides a base on which to construct higher-level conversations oriented to particular ser-

³Note that UDP does not provide the stable addressing or multicast required for this application. Nor does it define a mechanism that provides both request-response and datagram interaction through the same mechanism, as provided by VMTP.

⁴The message transaction can be regarded as a degenerate form of virtual circuit in which the request is a combined "open" and "user-data" packet and the response is a combined "user-data" and "close" packet.

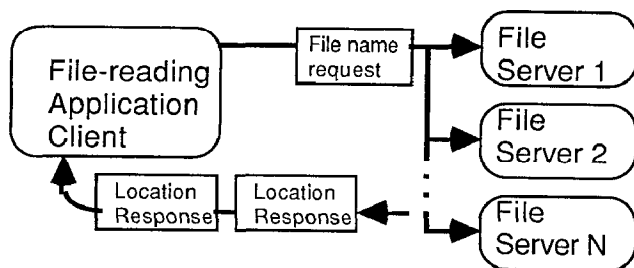


Figure 3: VMTP Group Message Transaction: File Name Query

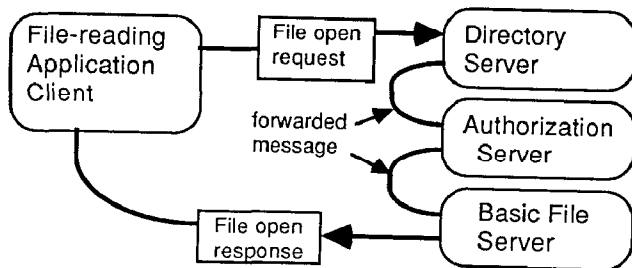


Figure 4: Forwarded Message Session: Authorized File Open

vice protocols, the most common and performance-critical example being open file connections. It also supports application-specific conversations. The next section describes the flexibility available within the framework of the basic message transaction.

2.2 Variants on the Basic Message Transaction

VMTP provides three variants on this basic VMTP session to widen its applicability and efficiency.

In the first variant, the *group message transaction*, the client sends or *multicasts* to a *group* of server entities and may receive multiple responses. The message transaction is terminated by the client initiating a new message transaction. Normally, a client only retransmits until the first response is received. A group message transaction session is illustrated in Figure 3 by a client multicasting to a group of file servers to locate a file. In this figure, two file servers respond, indicating they both have a copy of the named file.

In the second variant, the client sends the request as a *datagram* with an indication that no response is expected. The request may be sent reliably or not (acknowledged or unacknowledged). In the former case, the client may not issue another transaction until the datagram is acknowledged. In the latter case, the datagram may be discarded by VMTP if the client issues a new message transaction before the datagram is received.

In the final variant, the request message may be *forwarded* to another server in which case the other server responds directly to the client, as illustrated in Figure 4. In this figure, the client sends an open request to a directory server which, after locating the specific file, forwards the request to the authorization server, which after authorizing access, forwards the request to the basic file server. The latter server can refuse to accept open file requests unless they are forwarded to it by the authorization server. The basic file server replies directly to the client. Viewed as a series of remote procedure calls, the sequence logically behaves as though the directory service called the authentication service which called the basic file service. However, the return from the basic file server is optimized by going directly back to the client. In general, forwarding is used with remote procedure calls to optimize for the case in which the last statement of a procedure is to call another procedure and return that procedure's return value. This optimization is analogous to tail-recursion elimination in compiler optimization except, in the case of forwarding, one is saving network packets and hundreds, if not thousands, of instructions, not just a few instructions. A forwarded transaction is the same as multiple transactions by the client, first to the original server and then to the "forwarder" except

that:

- The client need not support the forwarding of messages, i.e. know that a message needs to be rerouted.
- Forwarding is faster.
- The forwarder of the message can confer some authorization on the message, as suggested in the above example.

These variants can be combined in various forms, e.g. datagram group message transaction. Most of these forms have been used extensively in the V distributed operating system[4,8,7]. The interested reader is referred to the cited references for further example uses.

There are several advantages to the basic session structure and its variants.

Higher Level Conversation Support: The basic VMTP session provides an efficient and reliable base for implementing multiple types of higher level conversations, one example being open file connections. It also avoids redundancy, overhead and restrictiveness of using connections provided directly as part of the transport level.

Minimal Packet Exchange: The basic message transaction provides the minimal 2-packet exchange for simple connectionless interactions such as file query, get time and simple remote procedure call.⁵ In contrast, a virtual circuit protocol such as TCP would require 8 packets for most implementations (although 6 is adequate in theory) unless the message transaction uses an existing connection.

Ease of Use: Providing the multicast, datagram and forwarding variants as part of the same protocol means that clients and servers that use multiple types of message transactions can share considerable common code for the handling the different cases. For instance, it is not unusual for a server in the V distributed system to receive basic message transactions, group message transactions and datagram message transactions. Since they are all supported by one protocol, the server uses a single, simple "receive" operation to wait for the next message transaction, which may be any of the variants.

Shared Communication Code: The variants are all simple modifications of the basic message transaction and therefore have considerable code in common. Thus, a VMTP implementation can support reliable request-response transactions, , multicast and datagrams at a lower space cost than providing each as a separate protocol. For instance, a datagram message transaction uses the normal VMTP transmission mechanism; it just does not wait for a response. Similarly, delivery of datagram requests is handled by the same mechanism as other types of requests. In general, these variants are simple and efficient to implement as extensions of the basic VMTP implementation.

A common alternative is to realize RPC support using a virtual circuit protocol. This approach provides limited advantages and imposes significant costs and restrictions. In particular, a virtual circuit provides: (1) lower-cost addressing once a circuit is established, (2) synchronization and streaming flow control, and (3) authentication on circuit setup that is amortized over use of the circuit. The benefit of small addresses is limited and seldom realized because it requires state in the network or internetwork that "understands" these smaller addresses. Thus, TCP, for example, sends the full Internet host addresses plus port identifiers of the source and destination in every packet. In subsequent sections, we argue that VMTP achieves efficient flow control, synchronization and authentication without virtual circuit support. The previous section pointed out the restrictions and costs that arise from this approach⁶.

2.3 Stable Addressing

The *stability* of VMTP addressing is essential in providing an adequate basis on which to build higher-level conversations, as described in Section 2.1. In particular, the client can use a server identifier repeatedly

⁵The protocol achieves this minimum when there are no errors, no retransmissions and the data for the request message and the response message each fit in a single packet. Otherwise, additional packets are required but the number of packets is still close to minimal.

⁶Note that a datagram facility in itself does not support the common case of remote procedure call and file read since it does not provide the pairing between request and response or reliability delivery. Thus, we do not consider datagram service as a reasonable alternative to message transactions at the transport level.

and be assured that the identifier either maps to the same server entity as it did on its previous use or else it is now invalid. Similarly, the server can compare the entity client identifier of the client with a previous client entity identifier and be assured that it is the same client if and only if the entity identifiers are the same. Finally, a server can test the validity of a client identifier. If invalid, the server can discard the high-level connection resources associated with that entity.⁷

A VMTP entity identifier is either strictly stable or T-stable.

Strictly Stable Identifiers A *strictly stable* identifier has the same binding or meaning forever once it is bound although it may appear invalid at times. Strictly stable entity identifiers are typically only used to identify a service such as the group of file servers, serving the same function as *well-known ports* in TCP/IP and XNS. In this example, the identifier refers to file service if any of the file servers are operational and otherwise appears invalid in the sense of “unbound”. Generally, strictly stable identifiers should be administratively bound and used sparingly. T-stable identifiers are used for all other entity naming, including instantiations of services such as a particular file server, processes to execute application programs, and groups of entities such as jobs and parallel programs that have a transient existence.

T-Stable Identifiers A *T-stable* identifier has at most one valid binding or meaning over a time interval T , but unlike a strictly stable identifier, can be rebound over time. A T-stable entity identifier is guaranteed not to be reused for at least T seconds after it becomes invalid. Thus, if the entity identifier is checked for validity more frequently than every T seconds, a change to the binding of the identifier will be detected. (VMTP provides a facility to probe the validity of an entity identifier.)

The use of T-stable identifiers provides an infinite supply of identifiers over time with T-stability limiting the confusion that reusing these identifiers can produce. T-stable identifiers are also useful in duplicate detection, as described in the next section. In contrast, the use of only strictly stable identifiers would require a very large identifier space since (for example) every process creation would permanently consume an identifier. VMTP uses a relatively large (64-bit) entity identifier, however, the structuring of this identifier for efficient mapping to network addresses reduces the effective number of identifiers, as described below.

Implementing T-Stability T-stability may be implemented following the scheme used previously in Thoth[3]. Basically, an entity identifier is structured as a *index* plus a *generation* field. The maximum number of entities allocated at any time is limited to I , the number of indices. A list of the last used generation for each index is stored in LRU order. To allocate a new identifier, the next generation associated with the next free index in the list is used. For instance, if the least recently used index is 3 and the last generation used with index 3 was 42, then the new identifier will be (3,43). This scheme guarantees a minimum recycle time of at least G allocations where G is the number of generations, occurring when a single index is free. However, with a sparsely used identifier space, the expected recycle time is much greater. It also uses a modest amount of storage, an array of I elements, each capable of holding 0 to $G-1$. In contrast to this scheme, implementing strict LRU order use of identifiers would require in the worst case that one store the entire set of free identifiers in LRU order. In addition, a scheme that simply increments through the identifier space can have 0 recycle time. That is, an identifier can become invalid just before it is chosen again as the next one to allocate.

For ease of allocation and mapping, each entity identifier is actually structured as two fields, as in V[8], a *logical host* portion and a *local identifier* portion, as shown in Figure 5. The index-generator structure may be imposed on each subfield. For instance, one can have 4 million index values with a 1024-word table storing the last generation for each

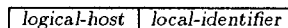


Figure 5: Entity Identifier Substructure

logical host identifier index of 22 bits. This table can be replicated across all hosts, with a new host consulting existing hosts to get a copy of the table and allocate a new logical host number.⁸ This division limits the number of logical hosts to 1024. Using a similar approach for the local identifier limits the number of entities to 1024 per logical host and thus a million in total. (Note that multiple logical hosts per physical host are allowed.)

The division of the local identifier into index and generator portion is a decision local to each host as is the allocation of new entity identifiers using a given logical host identifier. In fact, a host is free to use other techniques for guaranteeing a lower bound on the minimum recycle time of its entity identifiers.

Stability of entity identifiers is important for correct handling of delayed duplicates. As described in the next section, duplicates are detected provided

$$T_{stable} > 6 * T_{packet}$$

where T_{stable} is the time a T-stable identifier is stable and T_{packet} is the maximum packet lifetime. Using the IP time-to-live (TTL) field, T_{packet} can be limited to under 30 seconds or so (at least in theory) and T_{stable} time of greater than 3 minutes is easily achievable, given a realistic rate of crashing and process destruction.

These techniques limit the number of entities that can participate at any time. They also work best when the number of entities is significantly below the limit. To avoid going to larger identifiers or unreasonably restricting the number of entities using VMTP, we divide entities into entity domains.

Entity Domains An *entity domain* is a closed group of VMTP communicating entities such as a cluster of workstations sharing common network file service. Entities in two different domains can have the same identifier and only entities in the same domain may communicate directly via VMTP. However, an entity may participate in multiple entity domains simultaneously. For instance, a file server may serve multiple domains, having a separate entity identifier in the two domains.

Entity domains are intended to represent administrative and authorization domains. Entity domains are identified by a 32-bit entity domain identifier. This identifier is contained in each packet, allowing a host to identify packets within its domain(s) and ignore all others.

Entity identifiers are grouped into domains for several reasons.

- **Large number of VMTP entities:** The total number of VMTP entities can be very large by using a large number of entity domains even though the number of entities per domain is limited.
- **Small Number of Interacting Entities:** Domains reflect the expected use of VMTP, as a basis for closely interacting entities, such as in a distributed operating system. It does not appear feasible or desirable to operate the entire Internet as one large distributed operating system. For instance, the cost of dynamically assigning a new logical host identifier can increase with increasing numbers of participating hosts.
- **Access control:** Domains can be made to serve as spheres of trust, authentication and security within the Internet. For instance, a set of entity domains can represent different mandatory security levels in a multi-level secure system. Entities at one security level have no way to address entities at another security level.

Note that an entity can participate in two or more domains simultaneously by having an entity identifier in each domain. Thus, for example, a multi-level secure file server could participate in the each security level domain it supported.

⁸One logical host number is reserved for use in booting when a host has not yet been allocated a logical host number. Thus, two hosts booting simultaneously collide and randomly delay to avoid interference.

⁷Note that the (internet-host-addr, local-port) pair used for sockets in the Internet architecture does not have the required stability property because an Internet address does not designate a host, it designates an internetwork connection point. A multi-homed host has multiple Internet addresses. A mobile host may have its Internet address change. Thus, a server could not be sure whether a packet was from the same client as previously or not using this form of identification. In fact, some Internet implementations ignore the Internet address and only use the port identifier for intra-machine packet routing.

2.4 Duplicate Suppression

Duplicate suppression becomes a significant problem for transport protocols in an internetwork environment where it is necessary to deal with long delayed duplicates. A VMTP implementation must be able to deal with duplicate request and duplicate response packets. VMTP provides several techniques for handling duplicates, allowing the most efficient technique to be used in each situation. We first discuss the detection of duplicate request packets and then consider duplicate response packets.

Duplicate Request Packets A request packet is identified by the (T-stable) source entity identifier and 32-bit transaction identifier. For a particular source entity identifier, transaction identifiers are assumed to be strictly increasing modulo 2^{32} within any time window of less than T seconds, what we call *T-monotonic*.⁹

To achieve T-monotonicity, a VMTP host maintains a record of the last used transaction identifier for each valid client identifier it implements. The transaction identifier is incremented on each new message transaction. The VMTP transaction identifier at 32 bits is large enough that the transaction identifier cannot wrap around in less than time T, assuming a reasonable maximum rate of message transactions¹⁰. The VMTP host also only uses T-stable entity identifiers as client identifiers¹¹. This allows it to use any transaction identifier with a new entity identifier because, since the entity identifier is T-stable, the new entity identifier has not been valid for at least time T, so the T-monotonicity property is trivially satisfied. On crash and reboot, a host discards all previous T-stable entity identifiers and allocates new T-stable identifiers, so again T-monotonicity is guaranteed¹². We need show that T-monotonicity is adequate for duplicate detection for a suitable value of T.

The receiver of a new message transaction allocates a transaction record for the new message transaction that records the source entity and transaction identifier plus other information. This record is stored for at least T_{rec} seconds and is then discarded if no further communication is received from the client. We require that

$$T_{rec} > T_{oldrequest}$$

where $T_{oldrequest}$ is the maximum age of a request relative to the last response the client received from the server identified (by a T-stable identifier) in the request. (The determination of $T_{oldrequest}$ is described later on.)

When a node receives a request packet and it has transaction record for that source, it compares the transaction identifier of the packet with that stored in the transaction record and determines accordingly whether the packet is part of a new transaction, part of the current transaction (possibly a retransmission of part of the current transaction) or a delayed duplicate. If it does not have a transaction record for that source entity, either it has not received a packet from that source for at least T_{rec} seconds or else it has been rebooted within the last T_{rec} seconds.

If the node has not been rebooted, it can safely assume this is not a duplicate since T_{rec} is greater than the maximum age of a request packet, as defined earlier. Thus, the packet is accepted as part of a new message transaction.

If the node has been rebooted, there are two cases to consider. First, if the destination entity identifier specified in the packet is a T-stable entity identifier, we know that the identifier is either newly valid in the last T_{stable} seconds or else invalid. Since the packet cannot be older than T_{stable} seconds (since T_{packet} is less than T_{stable}), either the destination identifier is invalid and so the packet is discarded or else

⁹Note that in comparing a transaction identifier k with another, the $2^{31} - 1$ values greater mod 2^{32} than k are considered larger and the next 2^{31} values are considered smaller with one equal to k . Thus, a transaction identifier m has wrapped around relative to k when $m = k \bmod 2^{31}$.

¹⁰For example, assuming a client generates a message transaction every 1 millisecond, it would take over 23 days for the transaction identifier to wrap around.

¹¹Strictly stable identifiers are primarily useful for identifying a service or group of related services so this is not a significant restriction.

¹²To continue to use the same T-stable identifiers after reboot, a host would have to retain the transaction identifier associated with each identifier plus ensure that the rest of the system did not detect these identifiers as being invalid or unbound during the time between crash and reboot.

the packet is not a duplicate. It remains to consider the case of the destination entity identifier being a strictly stable identifier.

VMTP does not provide automatic duplicate detection when the destination entity identifier is strictly stable and the host has been recently rebooted. There are several approaches the user level can take to handle this case. First, with some services, requests are *nilpotent*¹³ so a duplicate trivially causes no problem. A time service is a simple example: A delayed duplicate request for the time can be processed with no ill effect since it does not change the state of the time service and the response is discarded at the client (see below).

Second, a service protocol may require a three-way handshake above the VMTP level and thereby eliminate duplicates analogously to a transport-level 3-way handshake. For instance, in the V naming scheme[6], stable identifiers are only used with a client naming query to locate the server implementing a particular name. The client receives in response a T-stable identifier to request services from a specific server. Since the query is nilpotent with respect to the server, a duplicate causes no problem when not detected. Since other service requests use T-stable identifiers, duplicate detection is handled as described earlier. Other forms of nilpotent requests and server-level duplicate detection are possible.

Also, VMTP allows a server to hold a request and request retransmission of the request from the client before acting on the request. This is effectively a three-way handshake.

Finally, the server can ensure there is at least T_{rec} seconds between the crash and the time it begins accepting new request packets again after reboot.

The duplicate suppression problem also arises when a rebooted node has entities that join an entity group with a T-stable entity identifier less than T_{packet} seconds after reboot. A request addressed to the T-stable group identifier less than T_{rec} seconds after reboot is the same problem as when the entity group identifier is strictly stable and so can be handled similarly. Alternatively, entities on a rebooted node can defer joining such a group until at least T_{rec} seconds after reboot.

Thus, to guarantee correctness,

$$T_{stable} > T_{oldrequest}$$

for T-stable server entity identifiers, where $T_{oldrequest}$ is the maximum time between a server confirming to a client the validity of the server's entity identifier and the server receiving a request from this client. $T_{oldrequest}$ is calculated as follows.

There are two cases to consider: (1) the client is between message transactions with the server and; (2) the client is in the middle of a message transaction with the server. In the first case, a client holding a T-stable server identifier is assumed to probe the server at least every T_{probe} seconds to ensure the identifier is still valid. Let $T_{reliable}$ be the number of seconds that the client waits between when it sends the first probe request to the server and when it gives up on getting a response, including several retransmissions. Let $T_{retrans}$ be the number of seconds between the first transmission of a probe request and the last retransmission before the node gives up on getting a response. Clearly $T_{retrans} < T_{reliable}$. In the worst-case scenario, the client receives a probe response just under $T_{reliable}$ seconds after it sent the first probe request. Moreover, the server received the probe request almost immediately; it was the response that was delayed by $T_{reliable}$. The client then starts sending a request to the server T_{probe} seconds later with the last retransmission of that request occurring at $T_{retrans}$ later. This packet is then delayed for just short of T_{packet} in the network before delivery. Under these assumptions, the oldest request a server host should receive is

$$T_{oldrequest} = T_{reliable} + T_{probe} + T_{packet} + T_{retrans}$$

In the worst case for T-stable identifiers, the server dies right after the probe response is sent, maximizing the time for the server identifier to be reused. The second case is the same except that T_{probe} becomes the time between probes as part of a message transaction.

T_{packet} , the maximum packet lifetime, is specified by the network. $T_{reliable}$ and $T_{retrans}$ are determined by the expected propagation delay $T_{avgdelay}$ and the maximum number of retransmissions required for

¹³A request is *nilpotent* if it results in no "significant" change to the server state. An example of an "insignificant" change is an update to the number of requests processed.

reliable delivery, R . Typically, $T_{reliable} = R * T_{avgdelay}$ and $T_{retrans} = (R - 1) * T_{avgdelay}$. The client must set T_{probe} and T_{rec} so that $T_{rec} > T_{oldrequest}$ and $T_{stable} > T_{oldrequest}$. The VMTP implementation must ensure that

$$T_{stable} \gg T_{reliable} + T_{packet} + T_{retrans}$$

since large T_{probe} is safer and more efficient for clients and servers. This should not be a problem using 64-bits entity identifiers and the techniques described in the previous section.

Duplicate Response Packets A response packet is discarded if it does not match the transaction identifier associated with the client entity identifier to which it is delivered or the client identifier is invalid. The client identifier is always T-stable and transaction identifiers are T-monotonic, as described with request packets. A response packet specifying a valid client identifier and the correct transaction identifier cannot be old enough to represent a previous use of these identifiers if certain timing relations hold, as described below.

In the worst case, a client sends a request to a server which the server receives (the maximum) T_{packet} seconds later. The server then takes T_{rec} seconds to process the request¹⁴ and sends back a response which takes T_{packet} seconds to arrive back at the client machine. Thus, the maximum age of a response packet relative to when the last retransmission of the request was sent is

$$T_{oldresponse} = T_{rec} + 2 * T_{packet}$$

There are three cases to consider to show that duplicates are correctly detected.

First, the client is still alive when the old response arrives. Then, duplicates are correctly detected providing the recycle time for transaction identifiers,

$$T_{recycle} > T_{oldresponse}$$

This condition holds for 32-bit transaction identifiers reasonable message transaction rates and reasonable values for T_{packet} .

Second, the client may have died immediately after sending the request but without the client node crashing. If each entity using a given index continues on with the transaction identifier value used by the last entity identifier for that index, a new client with entity identifier i cannot reuse a transaction identifier used by an entity previously bound to i in less than $T_{recycle}$. So, again there is no problem for reasonable values of T_{packet} . (Also, if $T_{oldresponse} < T_{stable}$, there is no problem.) Finally, the client machine may have crashed immediately after sending the request and a new machine rebooted using the same logical host identifier. Thus, to guarantee correctness on the client machine after reboot, it must be the case that

$$T_{stable} > T_{rec} + 2 * T_{packet}$$

for the client entity identifiers. This can be reduced to $2 * T_{packet}$ if the rebooting host broadcasts a request to flush transaction records for that logical host before it starts using the new logical host.

These relations depend on the worst-case delivery time, the expected delivery time and the expected packet loss rate. If we assume that $T_{reliable} \approx T_{packet}$ and $T_{probe} \approx T_{packet}$, then the system must ensure

$$T_{stable} > 6 * T_{packet}$$

without flushing transaction records and $4 * T_{packet}$ otherwise. The assumption of $T_{reliable} \approx T_{packet}$ is reasonable since, in a local network, $T_{reliable} > T_{packet}$ whereas in an internetwork $T_{reliable} < T_{packet}$. Unless the implementation discriminates, it must pick a value T which is greater than either value in either environment. Note that if T_{packet} is quite small, we can pick $T_{reliable}$ and T_{probe} small too, since a network or internetwork with small T_{packet} cannot have a long propagation delay. The most demanding case arises when both the worst-case and expected case delay are large.

¹⁴VMTP discards the request after T_{rec} seconds preventing the server from responding after that point.

Multiple Responses to a Group Message Transaction When a client sends to an entity group, it may receive multiple responses. A record of each response is retained by VMTP until the message transaction is terminated so that duplicate responses can be discarded. A response packet group is a duplicate if it matches the server identifier of some previously received response within this transaction.

Idempotent Transactions The discussion so far has focused on significantly delayed duplicates. When a duplicate request arrives at a server before it has generated the response, the duplicate is easily detected and discarded by the VMTP module, using the transaction record. When a duplicate request arrives after the response has been generated but before timeout of the transaction record or another message transaction from the same client, the response must be retransmitted in case the original response was lost. Note that another message transaction from the same client implicitly acknowledges receipt of the previous response.

Normally, VMTP saves a copy of the response message until the transaction record times out. However, for efficiency, the server can specify that the transaction is *idempotent*¹⁵. In this case, a retransmission of the request after the response has been sent is passed on to the server as a new request, allowing it to provide the response message anew. (Note that a duplicate of the last request is the only event that causes the server to retransmit the response.)

Specifying a transaction as idempotent eliminates the overhead of the VMTP transport module making a copy of the response to save for retransmission. This can be a significant saving on a file server, where read requests are the dominant operation and the file server buffer pool makes the server-level regeneration of the response fairly inexpensive, especially given the infrequency with which they should arise. Making a copy of a large response doubles the processing cost of sending the response if experience with the V kernel is any guide.

In summary, VMTP employs a diversity of approaches to dealing with duplicates, allowing the most efficient technique to be used in each different situation. Thus, the common cases of duplicate suppression are handled without the overhead of a 3-way handshake, extra delays or extra copying. In contrast, TCP requires a 3-way handshake for each circuit setup and circuit tear-down to deal with delayed duplicates. In theory a minimal TCP request-response behavior would require 6 packets: the 3 packets for connection setup and 3 packets for connection shutdown with overlap of the request and response plus the request. In practice, 8 packets are required because few TCP implementation support receipt of data as part of circuit setup or transmission of data as part of circuit clearing. Adding this extra overhead to every file read or write is unacceptable.

2.5 Selective Retransmission

Earlier communication systems assumed that retransmissions were required primarily because of errors in transmission. However, communication hardware has evolved to provide lower error rates and higher speeds so retransmissions are now more frequently due to overruns, either in the destination host or intermediate gateways and packet switches. The overrun problem is accentuated by several factors. First, a server such as a file server can suffer from overrun because it can be the focus of several clients' transmissions at the same time. The frequency with which this occurs rises with the load on the server. Second, a client can suffer packet overrun in receiving a multi-packet transmission from a server because the server may be faster hardware than the client, i.e. the shared resource is made more powerful because it must serve multiple clients. Third, bridges and gateways interconnecting high-speed networks usually cannot handle simultaneous traffic bursts from their connecting networks, causing packet drop or overrun at these intermediate hops. Finally, many hosts find it easier and more efficient in processing cost to send a large amount of data as a single blast of packets, transmitted as fast as the network allows, rather than pacing the flow.

¹⁵By *idempotent* transaction, we mean that processing multiple copies of the associated request received consecutively from the same client has the same effect as processing a single request, both on the state of the server and the response that is returned.

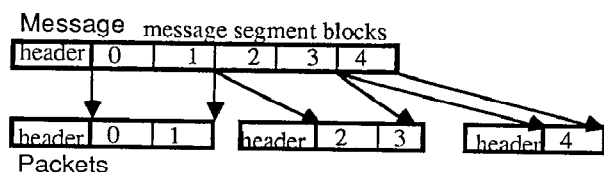


Figure 6: Packetizing a VMTP Message

When packets are lost due to overruns, systematic errors are more likely. For instance, the receiver may drop every 4th packet. If a sequence of packets are retransmitted at the same speed, the same packets will be dropped again.

These characteristics make selective retransmission attractive. For instance, if a group of N packets are transmitted at a speed that slightly overruns the receiver or some intermediate node, every k th packet may be dropped. Selectively retransmitting these dropped packets, possibly with a slightly longer inter-packet gap is significantly more efficient than retransmitting from the first dropped packet. A selective retransmission mechanism also gives a greater indication to the sender of whether overrunning is the problem and how severe the problem is. For instance, if the selected acknowledgement indicates that every k th packet was dropped, overrunning is the likely cause, with the value of k indicating degree of speed mismatch.

A VMTP message consists of a 76-byte header plus up to 16 kilobytes of *segment data*. Message segment data is divided into 512-byte segment blocks.¹⁶ A message is packetized into a *packet group*, a sequence of packets each containing a copy of the VMTP header plus zero or more blocks per packet. A 32-bit bitmask field indicates the segment blocks in each packet. Figure 6 illustrates the packetizing of a 2.5 kilobyte message for a network, such as the Ethernet where the maximum packet size is large enough for 1 kilobyte of data plus 76-byte header plus network and internetwork header space. Note that since the header is duplicated in every packet, packets can be sent in any order plus receipt of any packet in a packet group allows the recipient to respond with a bitmask indicating the portions of the message that were received.

The bitmask provides a simple, fixed length way of specifying which packets were received as well as indicating the position of a packet within the packet stream. The combination of packet groups and a (re)transmission mask provide a flexible selective retransmission facility with no variable length header or acknowledge information. Also, the sender can efficiently recover from slightly overrunning the receiver or intermediate network nodes and is provided with information that aids in adjusting the transmission rate to reduce loss.

2.6 Packet Group-based Flow Control

VMTP uses packet group-based flow control: the transmitter sends a group of packets of at most 16 kilobytes of data as one operation, one VMTP message. The receiver accepts and acknowledges this packet group as a unit before further data is exchanged. A packet group is typically “blasted” at full speed. Selective acknowledgement and retransmission in VMTP provides a means of dynamically detecting when the transmission rate is too high. An implementation should increase the inter-packet delay until packets are not being dropped due to overruns. In addition, one can refine the implementation with estimates of appropriate inter-packet and inter-packet group intervals to minimize packet loss due to overruns, as suggested by Clark et al.[11].

Amounts of data larger than 16 kilobytes can be transferred in one of two ways in VMTP. For one, the VMTP user can construct a high-level conversation, such as an open file connection, and read or write arbitrary amounts of data as a sequence of message transactions within this conversation. Second, the server can call back to a *memory server*

¹⁶The choice of 512 bytes is a compromise between local network packet sizes and packet radio packet sizes. Multiple 512-byte segment blocks are composed in an Ethernet packet, for instance. Packet radio with a packet size less than 588 bytes must use multiple packets to form a full-size VMTP packet.

for the client entity, thereby reading or writing the client’s memory to effect a large transfer. The latter is used to implement the interprocess copy operations, *CopyTo* and *CopyFrom* in the V kernel[4].

Packet group-based flow control is made feasible by the reduced cost of memory. It now is feasible for a file server to be prepared to accept up to 16 kilobytes of data off the network most of the time. However, a server can, if necessary, hold off the entire request, such as when its buffer space is exhausted. Similarly, it is feasible for a sending client to transmit 16 kilobytes as one blast as well as receive 16 kilobytes as a response.

The packet group approach simplifies the protocol and provides an efficient flow control mechanism for network file access behavior. This claim is supported by considering conventional flow control.

The conventional approach to flow control entails a *sliding window* with the objective of achieving continuous transmission of packets at the maximum delivery rate of the network and the maximum receive rate of the recipient. This assumes a continuous supply of packets to transmit, as in a file transfer. However, the sliding window approach can have a significant performance problem in the VMTP-targeted communication environment. For example, on a file write, flow control is required primarily because the file server may have limited buffer space and may not be able to dispose of data as fast as it is arriving. However, the file server only needs to stop the sender for a short period of time. In this situation with a sliding window protocol, the sender usually reaches the end of the window and then blocks waiting for authorization to continue. In a straight-forward implementation of the receiver, the receiver sends back further credit or advances the window very shortly thereafter. This results in the sender repeatedly filling the window, blocking for a short time and then, once the window is advanced slightly, sending another small amount of data and blocking again. Similarly, the receiver is repeatedly sending an acknowledgement or flow control signal on each packet. Thus, both ends incur maximal overhead for flow control. A worst-case of this problem can occur with TCP where the window size can become as small as 1 byte, known as the *silly window syndrome*[9].

The solution is to program the receiver so that the window is not advanced until it can be advanced by a significant amount. The net result is that packets are sent in bursts. This is efficient for the sender since a group of packets can be queued for transmission in one scheduling unit with the process suspended until the end of the packet group. Thus, sliding window protocols must, for efficiency, be forced to operate in sending packet groups, the same as VMTP. However, the packet group concept is explicitly represented in VMTP to force this efficient implementation. It has the added advantage of simplifying the handling of flow control as well as selective retransmission, as described above.

Sliding window protocols have another problem in the network shared file server environment. When a server is returning data to a client, considerable resources can be tied up if the client uses flow control to halt the server’s transmission. In particular, the client could neglect indefinitely to advance the window on data being returned in response to a read operation. The server would have buffers in the disk buffer pool as well as the communication module or kernel tied up indefinitely unless it chose to break the connection. In contrast, using flow control based on packet groups, a client must be prepared to accept an entire packet group at once. If a server returns data in units of packet groups, it can guarantee that its buffers will be tied up at worst for the maximum retransmission time for a packet group. It also forces clients to be structured to accept a full packet group at a time rather than incrementally process data on reception.

3 VMTP Protocol Details

VMTP assumes an underlying *delivery service* that provides end-to-end (best-efforts) datagram delivery, such as provided by IP and “raw” Ethernet. It is also designed to take advantage of datagram multicast facilities such as are available on the Ethernet and proposed for the Internet[5].

The packet layout, as shown in Figure 7, is logically structured as 4 portions:

- entity and transaction identification - including authentication

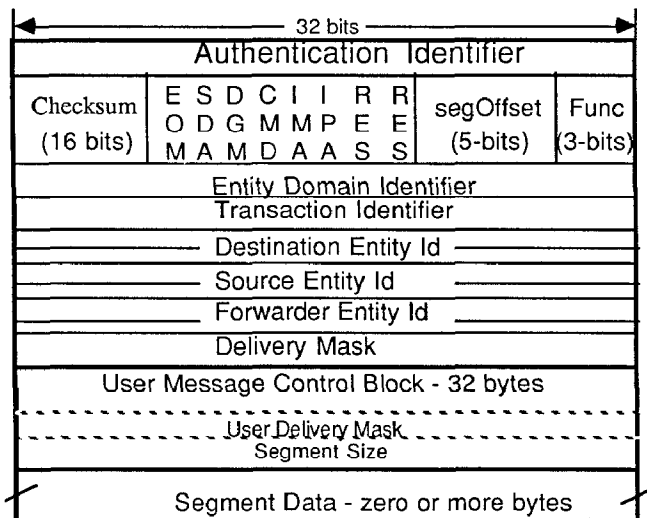


Figure 7: VMTP Packet Format

tion identifier, domain, source, destination, forwarder and transaction identifier.

- **packet group control** - including checksum, control flags, segment offset, function code and delivery mask.
- **user message control block** - system flags, user data and segment size.
- **segment data** - a maximum of 16 kilobytes of segment data, possibly further limited by the maximum packet size.

The use of most of these fields should be evident from the previous sections. We focus on a few fields that have not been discussed.

The *authentication identifier* field is used in one of two modes depending on whether messages are encrypted or not. If not encrypted, the authentication identifier indicates the authentication of the sending entity, such as a user account within the entity domain. If encrypted, the authentication identifier is used to route the encrypted message to the right decryption key within the receiving node, the same as the *conversation key* used by Birrell[2]. In fact, we propose following Birrell's design for secure VMTP communication. As pointed out elsewhere[8], this design extends to handle multicast.

In packet group control, the 8 bits of control flags determine aspects of delivery, as described below.

EOM End Of Message - last packet transmitted as part of this packet group that constitutes a message, else 0. This is a hint to the receiver but is only appropriate to utilize when the network does not (or very infrequently) reorders packets during delivery.

SDA Segment Data Appended - segment data is appended in the packet group of length segmentSize, or 16 kilobytes, whichever is less.

DGM Datagram request - If a request message, do not wait for reply, or retransmit. If a reply message, treat this transaction as idempotent.

CMD Conditional Message Delivery - only deliver the request or response if the receiving entity is waiting for it at the time of delivery, otherwise drop the message.

IMA Immediate Message Acknowledgement - Acknowledge packet group immediately after received. If a request, send back a request-ack packet as soon as the request packet group is received, independent of the response being available. If a response, send back a response-ack packet as soon as the response packet group is received.

IPA Immediate Packet Acknowledge - as with IMA except the acknowledgement is sent immediately on receipt of this packet. This allows the sender to get intermediate acknowledgements during the transmission of a packet group.

RES 2 control bits reserved for future use.

These control flags do not significantly complicate the protocol implementation because they do not introduce new protocol states (unlike TCP control bits) and need to be examined only in a small number of places in the packet handling code. For example, both *IPA* and *IMA* are examined after processing the packet and simply cause the implementation to generate an acknowledgement packet if set.

The *segOffset* indicates the offset of the beginning of the segment data (in segment blocks) within the message segment data. The *delivery mask* field is the 32-bit bitmask indicating the message segment blocks being transmitted as part of the packet group.

The 3-bit *function code* field indicates the packet type, one of:

- 000 Request
- 001 Response
- 010 Forward-request
- 011 Forward-response
- 100 Request-ack
- 101 Response-ack
- 110 Probe-request
- 111 Probe-response

The User Message Control Block (MCB) portion is specified by the VMTP user. This area is included in the VMTP header for several reasons. First, we observe that a large number of request and response messages are quite short. In our experience with the V distributed system, most remote procedure calls have less than 28 bytes of parameters. Examples include file open, close, query as well as read and write operations with a small amount of data. (The first 4 bytes are used normally to specify the procedure.) The user data field provides protocol support for combining the transaction record and buffer area for this size of data. Second, the user data area in the header is transmitted in each request and response packet that constitutes a packet group. Thus, the receiver is guaranteed to receive this data if any packets in a packet group are received. This supports idempotency and user-level handling of retransmissions, as appears desirable for efficient network file access. It also supports out-of-order reception. Finally, the user data area provides protocol support for a simple scatter-gather facility. For instance, in a file write operation, the parameters for the write go into the 32-byte user data field. The data to be written, typically located elsewhere, is transmitted as part of the data segment. We view the user data area in the header as analogous to general-purpose registers in a CPU. In a CPU, the registers are a small, fixed-sized area available to the user with various performance benefits. The same comment holds for the VMTP user data area. Similarly, compiler writers recognize that making use of registers is difficult and complicates the compiler. However, the performance benefits warrant the inconvenience. The user delivery mask and segment size are subfields of the user message control block. The user is free to use these fields for other data if they are not being used in this capacity, i.e. no segment data.

Thus, the basic VMTP header (without data segment) is 76 bytes. Including segment data, a VMTP message can be as large as 76 bytes plus 16 kilobytes for segment data. However, the actual transmission size includes the IP header, network header plus the 76-byte header replicated in each packet. (The minimal IP-based VMTP packet is 96 bytes plus network header, e.g. 14 bytes in the case of Ethernet.)

4 Current Status and Performance

VMTP is currently being reviewed as a candidate for the Internet request-response protocol by the Internet End-to-end Task Force. As part of the technical review, VMTP is also being implemented in a

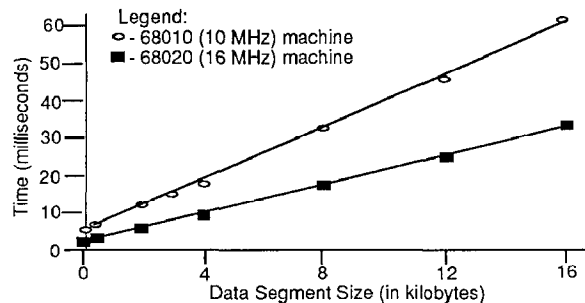


Figure 8: VMTP Performance on Raw Ethernet

version of the V kernel as a refinement to the original V kernel IPC protocol. At the time of writing, the protocol software has been implemented in a preliminary form. The VMTP code takes about 1700 lines of C code resulting in 7.8 kilobytes of code and initialized data on a Motorola 680X0. (This figure does not count the basic Ethernet driver or IP support.)

Figure 8 gives the performance of VMTP running between two Motorola 68020-based machines¹⁷ and two Motorola 68010-based machines¹⁸ using raw Ethernet interconnection. (This implementation is a modified version of the V kernel using VMTP for network interprocess communication.) In this prototype implementation, each packet contains at most one 512-byte segment block, rather than 1024 bytes. Thus, for example, transmission of 16 kilobytes used 33 packets, rather than the 17 packets (1 for the request and 16 for the response) that the final protocol implementation will support.

Figure 8 shows that, as expected, the time for transmission grows linearly with size of data segment. With 16 kilobyte requests, the data rate is roughly 461 kilobytes per second or 3.69 megabits per second, a reasonable performance level compared to most TCP implementations, especially given that we are sending double the number of packets required. Packet loss and network load would of course degrade this performance, although these have not been an issue in our experience to date. With the 32-byte user data area in the packet header, exchanging small amounts of data is very efficient, 2.5 milliseconds round trip on the 68020 machines.

Our experimental setting is ideal for VMTP, providing low delay, high data rate and low error rate, as is typical of local networks. We plan to study the performance of the protocol further with networks and internetworks with longer propagation delay. We also plan to experiment with rate control techniques in both local area and wide-area network settings.

5 Related Work

VMTP is a refined version of the V inter-kernel protocol[4,8]. The refinements include: selective retransmission, larger transaction identifier, entity domain identifier and reduction in the number of packet types. A number of fields, particularly the transaction identifier field and the entity identifiers, have been extended to accommodate Internet parameters.

VMTP is similar to the transport level protocol described by Birrell and Nelson[1] for their remote procedure call system. However, their design does not provide for multi-packet messages with single acknowledgements. They also do not support multicast or datagram.

The VMTP timeout mechanism on transaction records is similar to that used in the Δt protocol of the Fletcher and Watson[12]. In fact, their calculations for Δt are analogous to our timer calculations. However, VMTP differs significantly from their work in the use of T-stable identifiers plus some specialization for request-response interaction and packet groups. They focus on independent messages using sliding window flow control.

The packet group mechanism with optional rate control is similar to the scheme used in the Netblt protocol proposed by Clark et al.[11].

¹⁷SUN 3's with 16.6 MHz clock rate.

¹⁸SUN 2's with 10 MHz clock rate.

However, Netblt is oriented towards bulk data transfer, not request-response interaction, and uses a virtual circuit connection. It is clearly not designed to support remote procedure calls. It also does not consider multicast.

6 Concluding Remarks

VMTP is a departure from current standard transport-level protocols in several ways. First, it uses a message transaction model of session, designed to support remote procedure call, multicast and real-time datagram service with optimization for supporting higher-level connections such as open file connections.

Second, rather than implementing connections, it provides message transactions and T-stable identifiers as basic facilities on which the higher-level can construct conversations.

Third, VMTP provides several different mechanisms for detecting and handling duplicates, ranging from 3-way handshakes at the VMTP or user level to relying on the T-stability of the addressing to exploiting the idempotency of common case message transactions such as file page read. In this way, a duplicate handling technique can be chosen for a given situation that is generally lower cost and delay than one fixed solution that is adequate for all cases.

Fourth, VMTP provides an inexpensive form of selective retransmission using the concept of packet groups plus a bitmask for indicating packets delivered. The selective transmission facility provides an efficient means of recovery from overruns, recognizing that the major source of packet loss is overruns, not transmission errors.

VMTP implements a simple form of flow control based on packet groups with optional rate control on the inter-packet gap within a packet group. We argue that this design provides an efficient flow control behavior for the situation in which the sender is slightly faster than the receiver, a common case with client to file server, and file server to client communication. Implementations of sliding window protocols need to mimic this behavior for efficiency. VMTP provides it directly. The selective transmission facility provides information for adjusting the "blast" rate to control loss due to overruns. That is, selective retransmission patterns that suggest overrun are recognized, causing the rate control to increase the inter-packet gap.

As a consequence of this design, performance-critical operations such as a network file page read can be handled with the minimal two packet exchange.

The purpose of this paper is to put forward the design ideas, not the protocol specification itself. It is likely that further modifications to the protocol will take place. In general, VMTP reflects a reappraisal of communication techniques motivated by significant changes in the use of communication systems and the underlying hardware base. We anticipate more changes of this nature in the future.

7 Acknowledgements

The Internet End-to-end Task Force, chaired by Bob Braden, has been invaluable in discussions on VMTP, leading to significant improvements in the design. I am also grateful to Dave Clark of MIT who has been a major participant in these discussions. Various members of the Distributed Systems Group have also contributed to the design and presentation, most notably Tim Mann, Lance Berc, Michael Stumm and Steve Deering. This work was sponsored in part by the Defense Advanced Research Projects Agency under contract N00039-83-K-0431 and by Digital Equipment Corporation.

References

- [1] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems*, 2(1), February 1984.
- [2] A.D. Birrell. Secure communication using remote procedure calls. *ACM Trans. on Computer Systems*, 3(1), February 1985.
- [3] D.R. Cheriton. *The Thoth System: Multi-process Structuring and Portability*. American Elsevier, 1982.

- [4] D.R. Cheriton. The V kernel: a software base for distributed systems. *IEEE Software*, 1(2), April 1984.
- [5] D.R. Cheriton and S.E. Deering. Host groups: a multicast extension for datagram internetworks. In *9th Data Communication Symposium*, IEEE Computer Society and ACM SIGCOMM, September 1985.
- [6] D.R. Cheriton and T. Mann. *A Decentralized Naming Facility*. Technical Report STAN-CS-86-1098, Computer Science Department, Stanford University, April 1986. Also available as CSL-TR-86-298.
- [7] D.R. Cheriton and M. Stumm. Multi-satellite star: structuring parallel computations for a workstation cluster. *Distributed Computing*, 1986. To appear.
- [8] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Trans. on Computer Systems*, 3(2), May 1985.
- [9] D.D. Clark. *Window and Acknowledgement Strategy in TCP*. Technical Report RFC 813, Defense Advanced Research Projects Agency, 1982.
- [10] E. Cooper. Replicated procedure call. In *10th Symp. on Operating Systems Principles*, pages 63–78, December 1985. Also published as *Operating Systems Review* 19(5), 1985.
- [11] M. Lambert D.D. Clark and L. Zhang. *NETBLT: A Bulk Data Transfer Protocol*. Technical Report RFC 969, Defense Advanced Research Projects Agency, 1985.
- [12] J.G. Fletcher and R.W. Watson. Mechanism for a reliable timer-based protocol. *Computer Networks*, 2:271–290, 1978.
- [13] D.P. Reed J.H. Saltzer and D.D. Clark. End-to-end arguments in system design. *ACM Trans. on Computer Systems*, 2(4):277–288, November 1984.